Genetic Programming over Spark for Higgs Boson Classification

Hmida Hmida, Sana Ben Hamida, Amel Borgi, Marta Rukoz

Genetic Programming over Spark for Higgs Boson Classification

Hmida Hmida[1,2], Sana Ben Hamida[2], Amel Borgi[1], and Marta Rukoz[2]

[1] Université de Tunis El Manar, Faculté des Sciences de Tunis, LR11ES14 LIPAH, 2092, Tunis, Tunisie
[2] Université Paris Dauphine, PSL Research University, CNRS, UMR[7243], LAMSADE, 75016 Paris, France

Abstract. With the growing number of available databases having a very large number of records, existing knowledge discovery tools need to be adapted to this shift and new tools need to be created. Genetic Programming (GP) has been proven as an efficient algorithm in particular for classification problems. Notwithstanding, GP is impaired with its computing cost that is more acute with large datasets. This paper, presents how an existing GP implementation (DEAP) can be adapted by distributing evaluations on a Spark cluster. Then, an additional sampling step is applied to fit tiny clusters. Experiments are accomplished on Higgs Boson classification with different settings. They show the benefits of using Spark as parallelization technology for GP.

Keywords: Genetic Programming· Machine Learning· Spark · Large Dataset · Higgs Boson Classification

1 Introduction

Digital transformation that we witness in organizations and companies have generated a huge volume of data. High storage capacities facilitated this phenomenon and provided organizations with their own data lakes[3].

To discover hidden knowledge in this data, many Artificial Intelligence (AI) tools and techniques have been used such as Neural Networks, Decision Trees, etc. Evolutionary Algorithms (EA), and in particular Genetic Programming (GP) [14], are candidate solutions since they have shown satisfying results for a wide range of problems (classification, time series prediction, etc.). However, GP suffers from an overwhelming computational cost. This becomes more noticeable when the handled problem has a very large dataset as input. In fact, the evaluation step (see figure 1) is the Achilles heel of GP. It is at the origin of the increasing computational time. Therefore, any solution that applies GP to solve a large scale problem, has to focus on reducing the evaluation cost.

Summarily, mitigating this cost could be achieved by either parallelizing evaluations or reducing their number by means of dataset sampling, hardware ac-

[3] 'A data lake is a collection of storage instances of various data assets additional to the originating data sources.' (Source: Gartner)

celeration or distributed computing. Hadoop MapReduce[4] and Apache Spark[5] are Big Data tools that implement a new programming model over a distributed data storage architecture. They are *de facto* tools for data intensive applications. However, according to our knowledge, neither Spark libraries, such as Spark built-in library MLlib, nor existing *ad hoc* solutions do provide an implementation of GP adapted to Spark, which hinders its use in Big Data frameworks. Moreover, recent machine learning problems more often than not need very high computing power and resources in order to make a solution in a reasonable time. The challenge of running GP on Spark becomes more defying when only Small clusters are available.

This work outlines how we ported an existing GP implementation to Spark context in order to take the most of its proven potential. We apply this solution to the Higgs Boson classification problem (see section 2.4) and study the effect of varying some GP parameters on learning performance and time. Additionally, we include a sampling algorithm to GP and test it in the same environment configuration used with the whole dataset. We discuss the contribution of this sampling method to the learning process and the effect of varying number of generations, population size and sample size on training time and classifier performance.

The remainder of this paper is organized as follows. Section 2 gives an introduction to GP basics. It presents Spark and recent works that combine these two concepts. Section 3, exposes how we adapted an existing GP implementation to comply with Spark environment. Then, we show why and how a sampling phase is added to GP. Details about experimental design followed by the obtained results are discussed respectively in Sections 4 and 5. Finally, we end with some concluding remarks and perspectives.

2 Background and Related works

2.1 Genetic Programming (GP)

In the standard GP, the population is composed of tree-based individuals very close to Lisp programs. It performs the common steps of any EA that are:

1. Randomly create a population of individuals where tree nodes are taken from a given function and terminal sets. Then evaluate their fitness value by executing each program tree against the training set.
2. According to a fixed probability, individuals are crossed and/or mutated to create new offspring individuals.
3. New solutions are evaluated and a new population is made up by selecting best individuals from parents and offspring according to their fitnesses.
4. Loop step 2 and 3 until a stop criteria is met.

The evaluation step is at the origin of the increase of the GP computing time. In fact, it executes all programs (individuals) as many times as the size of the

[4] https://hadoop.apache.org
[5] https://spark.apache.org

training set. In a classification problem, an individual is a program tree that represents a classification model. Figure 1 illustrates the evaluation of an individual against the training set. Its phenotype is $(if(IN3 > 0.3, IN3, IN0+IN1) < 0.6)$ and is represented by the tree in the same figure. It depicts a single iteration in the evaluation of a single individual within the population. The output is translated in a class prediction and is compared to the given value from the training set. The fitness value is computed, after looping on the whole training set, according to the fitness function adopted (error rate, true positive rate, etc.).

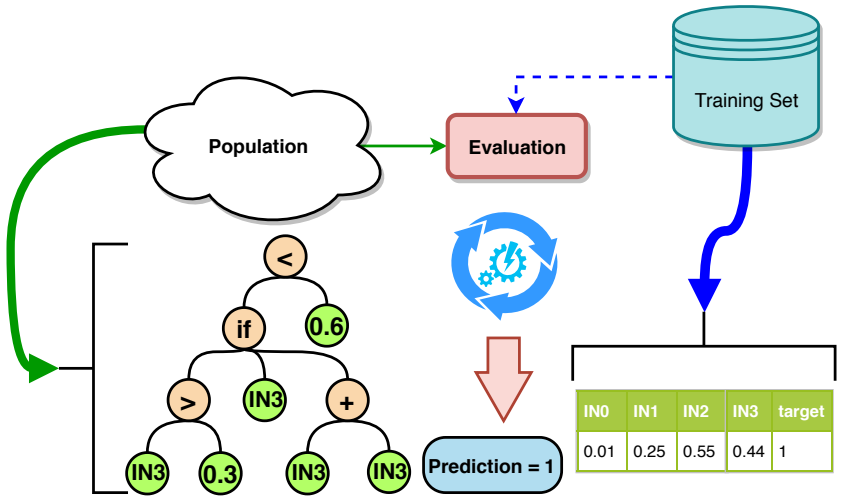


Fig. 1. GP Evaluation.

2.2 Spark and MapReduce

MapReduce is a parallel programming model introduced by Dean et al. in [5] and made popular with its implementation Apache Hadoop. The main idea of this model is that moving computation is cheaper than moving data. On a cluster, data is distributed over nodes and processing instructions are distributed and performed by nodes on local data. This phase is known as the Map phase. Finally, at the Reduce phase, some aggregation is performed to produce the final output. Hadoop Distributed Files System (HDFS) ensures data partitioning and replication for MapReduce. However, it needs many serialization operations and disk accesses. In addition to that, Hadoop MapReduce does not support iterative algorithms which is the case for EA.

Apache Spark is one of many frameworks intended to neutralize the limitations of MapReduce while keeping its scalability, data locality and fault tolerance. It is up to 100 times faster than MapReduce owing to in-memory computing. The keystone of Spark is the Resilient Distributed Datasets (RDD) [19]. An RDD is a typed cacheable and immutable parallel data structure. Operations on RDDs are of two types: transformations (*map. filter, join, ...*) and actions (*count, reduce, reduceByKey, ...*). Spark DAGScheduler, computes an

optimized Directed Acyclic Graph exploiting lazy evaluation and data locality. Spark is compatible with different cluster managers (Built-in Standalone, Hadoop YARN, Mesos and Kubernetes) [13].

2.3 Previous works

Recently, several works deal with parallelizing EA in distributed environments. Rong-Zhi Qi et al. [17] and Padaruru et al. [15] apply Genetic Algorithms (GA) on software test suite generation where input data is a binary code. Both give a Spark based implementation that parallelizes fitness evaluation and genetic operators. In Chávez et al. [4], the well-known EA library ECJ is modified in order to use MapReduce for fitness evaluations. This new tool is tested using GA to resolve a face recognition problem over around 50MB of data. Only time measure was considered in this work. In Peralta et al. [16], the MapReduce model is applied to implement a GA that preprocesses big datasets (up to 67 millions instances). It applies an evolutionary feature selection, in a map phase, to create a vector per mapper on top of disjoint subsets from the original dataset. The reducer aggregates the previously created vectors. Funika et al. [7] implement an 'Evaluation Service' that can be solicited through a REST API. It is not an implementation of a specific EA algorithm but rather an outsourcing of the evaluation. This service can be used by any algorithm that requires the evaluation of an expression over a given dataset. They tested this service for 3 different expressions on datasets varying from 1MB to 1024MB. Al-Madi et al. [1] present a full GP implementation based on MapReduce. It relies on creating a mapper for each individual by storing the population on HDFS. Each mapper, calculates the fitness value for the involved individual. Then, the reducer collects the population and performs selection, crossover and mutation to create the new population. The focus is on increasing population size (until 50000) and only with small datasets classification problems.

Our proposal is inspired by all these works and mostly by Chávez et al. [4] where the authors have integrated the MapReduce model to the ECJ library in order to run EA on a hadoop cluster. Their goal was to evolve very large populations (up to 3 million individuals), which was achieved by using the checkpointing feature and serializing individuals in an HDFS file. This work is focused on population and did not study the use of massive datasets. Our proposal is rather concerned with training dataset. We both give a transformation of an existing tool, but our solution uses a different underlying infrastructure which is Spark engine and handles a big dataset to assess the advantages of distributing GP evaluation. Another difference is the EA algorithm applied. While Chávez used GA, we use standard tree-based GP. Finally, we extend this solution with a sampling technique.

2.4 HIGGS dataset

A Higgs or Z Boson is a heavy state of matter resulting from a small fraction of the proton collisions at the Large Hadron Collider [3].

From the machine learning point of view, the problem can be formally cast into a binary classification problem. The task is to classify events as a signal (event of interest) or a background. Baldi et al. [2] published for benchmarking machine-learning classification algorithms a big dataset of simulated Higgs Bosons that contains 11 million simulated collision events [10].

In this work, we propose to handle the whole dataset, which is a big challenge when using EA. Table 1 summarizes the main characteristics of the dataset.

Table 1. Higgs dataset composition.

Total of events	11 millions
Number of Attributes	28 real-valued (21 low-level and 7 high-level variables)
Percentage of signals	53%
Training set size	10.5 millions events
Test set size	500K events

3 Porting DEAP to Spark

DEAP (Distributed Evolutionary Algorithms in Python) [6] is presented as a rapid prototyping and testing framework that supports parallelization and multiprocessing. It implements several EA: GA, Evolution Strategies (ES) and GP. We decided to use this framework for the following reasons: (1) it is a Python package which is one of the 3 languages supported by Spark, (2) it implements standard GP with tree based representation and (3) it is distributed ready. The third point means that DEAP is structured in a way that facilitates distribution of computing tasks. DEAP is not natively compliant with Spark and do not use any of its parallel data structures (RDDs, DataFrames or DataSets).

To adapt DEAP for a parallel computing engine, the usual method is to replace the map method, of the Toolbox class, by a code that calls the desired parallelized operation. Unfortunately, Spark has a constraint that prohibits nesting RDDs. To evaluate an individual, we need to access our dataset stored in an RDD. Consequently, this solution is not feasible. The following paragraph shows how we transformed DEAP to benefit from Spark RDDs.

3.1 Implementation model

From recent works described in 2.3, two main scenarios for distributing any EA, and in particular GP, can be laid out:

1. Distribute the whole GP process: each mapper is an independent run that uses local data. It is very close to the co-evolution scheme or *island* model. By the end of this scenario, an aggregation is required to obtain the final solution. This aggregation is run on the driver program. For example, aggregation can be made through a voting using the best individual of each population in a classification problem. Another way is to take the best individual after a test on the whole training set. This solution needs more resources on the cluster.

2. Distribute population: this is suitable when a single population is evolved. Parallelization can be reached by partitioning the population using RDDs. Therefore, individuals are distributed on nodes, and each node processes the local individuals in spite of the total population. To proceed the following phase of the GP (see 2.1), the output of the previous phase is collected as a new RDD. It is composed of the modified individuals (RDDs are immutable).

The evaluation represents more than 80% of the total time cost in EA [9, 7]. We suggest to focus on evaluation which can be easily distributed on Spark cluster even with limited resources seeing that we do not need independent populations. Besides, for machine learning problems involving big training sets, data must be parallelized and then we cannot parallelize population. The fitness function considered in this problem is to maximize correct predictions. To adapt fitness computing to Spark, a first alternative is to replace the default evaluation function used in DEAP. Map operation on *TrainingRDD* does not use the individual but a function (func) representing the Genetic Program. This alternative makes as many reduce operations as the size of population. This generates an important cost even for small populations. A key rule in Spark optimization is to reduce the number of action operations with regard to transformations. Thus, we altered DEAP evaluation so it maps all the population on the *TrainingRDD*. This diminishes action calls to one call per generation. Figure 2 gives the global flowchart of the modified DEAP evaluation algorithm. First, training set is transformed into an RDD (*TrainingRDD*) from a file stored on HDFS and is cached (Figure 3, line 3). Then, at each generation, the population is evaluated against the training set by mapping their functions on *TrainingRDD* partitioned over worker nodes (Figure 3, line 9). To get fitness values, a reduce operation is performed (Figure 3, line 12). After, offspring can be generated on the driver node by applying mutation and crossover. The offspring replaces the old population and program loops until maximum generation number is reached.

3.2 Data sampling

GP is a costly algorithm and this is intensified with big datasets like HIGGS. For clusters with a small number of nodes, running GP for such problems remains of high-cost. Furthermore, in these datasets, redundancy is inescapable. Sampling is a very suitable technique to deal with this situation. Additionally, depending on the underlying algorithm, sampling may counter overfitting, enhance learning quality and allow large population or more generations per run.

For these reasons, we investigated the use of sampling with the previous implementation. In this work, we started by a simple sampling method which is Random Subset Selection (RSS) [8]. RSS combines simplicity with efficiency. Spark makes available two operations: *sample* and *takeSample*. They produce samples with an oscillating size around the specified target. For efficiency, we used *sample* which is an RDD transformation. *takeSample* is an action and cannot be optimized by Spark DAG Scheduler. The training sample is renewed at each generation before evaluating the population.

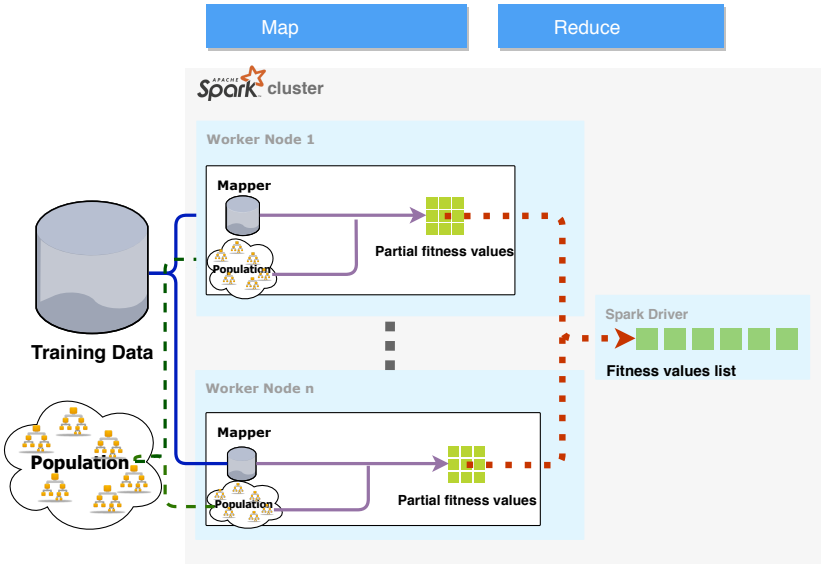


Fig. 2. Flowchart of the modified evaluation.

The commented code snippet in Figure 3 outlines the steps we made to adapt the DEAP standard GP.

4 Experimental settings

4.1 Framework

Software framework The details of used software are as follows: Spark version: 2.1.0, Hadoop version: 2.9.1, Resource Manager: YARN, Operating System: SMP Debian 4.9.130 and DEAP version: 1.2.2.

Spark cluster We used a tiny cluster composed of 4 worker nodes. Each node has a 16 core Intel Core processor at 2.397GHZ, 45GB of RAM and 1TB of HDFS storage space.

In the following experiments, Spark application is submitted to the cluster via spark-submit script. We used the same YARN directives that are optimized for the cluster size and the used dataset accordingly to guidelines in [12] for all the GP runs in order to neutralize the effect of this configuration on results.

On this cluster, 4 Spark executors are deployed per node. The number of available nodes does not allow us further investigation of hardware effect on GP performance.

4.2 GP settings

General settings Based on few runs, we set parameter values (Table 2) for GP. The process of tuning GP settings is beyond the scope of this work and have not been thoroughly studied.

```
from pyspark import SparkContext
sc = SparkContext(appName="DEAPSPARK")
TrainingRDD = sc.textFile("training.csv").cache() #parallelize training set
initGP()
while(generation<maxGeneration): # GP loop
    # serialize population and map it on training subset
    popFunctions = [toolbox.compile(ind) for ind in population]
    fitnessRDD = TrainingRDD.map(lambda line:\
        [getPrediction(func,line) for func in popFunctions])
    # compute final fitness using reduce
    fitnessValues =\
        fitnessRDD.reduce(lambda v1,v2:list(map(operator.add,v1,v2)))
    updatePopulationFitnesses(population, fitnessValues)
    # Select the next generation individuals
    offspring = select(population)
    # Apply genetic operators
    offspring = evolve(offspring, crossoverProb, mutationProb)
    population[:] = offspring
```

Fig. 3. Modified DEAP GP loop.

Terminal and function sets The terminal set includes 28 features of the benchmark Higgs dataset with a random constant. The function set includes basic arithmetic, comparison and logical operators reaching 11 functions (Table 3).

Table 2. GP settings.

Parameter	Value
Initialization	Ramped half and half
Tournament size	4
Tree limit	17
Crossover probability	0.9
Mutation probability	0.04
Generations and population size	61 different combinations

Table 3. GP terminal and function sets.

Function (node) set	
Arithmetic operators	$+, -, *, /$
Comparison operators	$<, >, =$
Logic operators	AND, OR, NOT
Other	IF (IF THEN ELSE)
Terminal set	
Higgs Features	28
Random Constants	1
Boolean values	True, False

5 Results and discussion

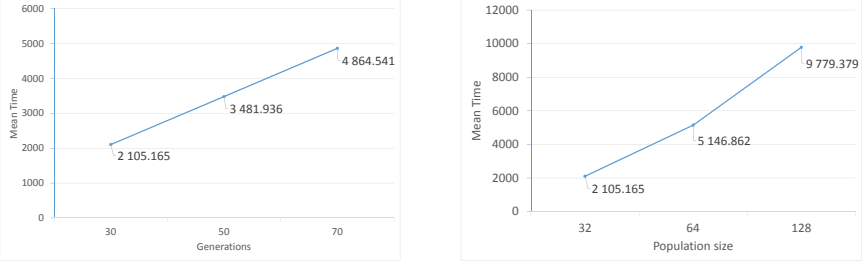
Since the main objective is to tackle computation cost of GP in supervised learning problems, the first recorded measure is learning time. It comprises the elapsed time from the initialization of the first population until the last generation. It does not include the time for evaluation against the test set.

Reducing time must not be at the expense of learning quality. Then, by the end of each run, the best individual based on the fitness function is evaluated on the test dataset. Results are recorded in a confusion matrix from which accuracy,

True Positive Rate (TPR) and False Positive Rate (FPR) are calculated. The objective function used with GP is the classification accuracy:

$$Accuracy = \frac{True\ Positives + True\ Negatives}{Total\ patterns} \quad (1)$$

We tried 61 different configurations obtained by varying population size, generation number and sample size. Each configuration is run 11 times. Then we compute the average learning time and the overall best individual performance metrics. By these experiments, we intend to trace the speed gain in learning time and how it reacts to population size and generations number changing.
The results are reported in Figure 4 in which 3 population sizes are tested (32, 64 and 128 with 30 generations) and 3 generations numbers (30, 50 and 70 with 32 individuals per population) with full dataset (FSS).

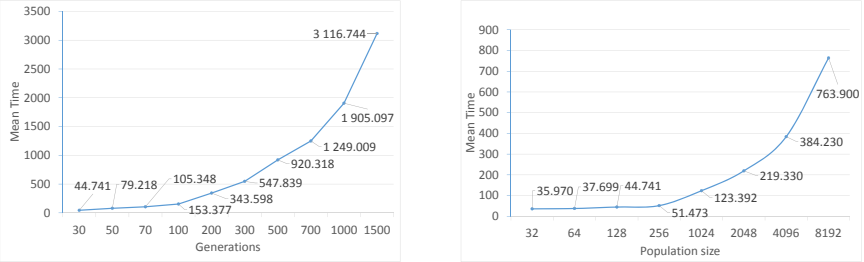


(a) Population Size = 32 (b) Generations = 30

Fig. 4. FSS Mean Time.

It is noticeable that parallelizing GP on Spark, facilitates its use for solving large classification problem like Higgs Boson classification. With only 4 nodes, a GP run takes on average 4864.541 seconds in 70 generation with 32 individuals and 9779.379 seconds in 30 generations with 128 individuals. A serial execution of GP on a single node takes more than 20 seconds to evaluate one individual against the total dataset which could take more than 44800 seconds for 70 generations with 32 individuals. Parallelizing evaluations under Spark achieves a speedup over nine times. This can be boosted by deploying more nodes on the cluster. Otherwise, learning time increases linearly with respect to the population size and generations number when using the same Spark settings.

Then, to allow using large populations and more generations without adding nodes, we injected RSS to the modified GP. We used the same settings but more population size values (from 32 to 8192) and generation numbers (from 30 to 1500) with a sample size fixed at 10000 instances. Results are exposed in Figure 5. The mean learning time for 1500 generations evolving 128 individuals is 3116.744 seconds and for 30 generations with 8192 individuals is 763.9 seconds. The two curves have the same pace. This is owing to the fact that evaluation is the predominant phase in GP. Also, with low values of population size and generations number, time curve is almost flat or with slight slope. It means that



(a) Population Size = 128 (b) Generations = 30

Fig. 5. RSS Mean Time.

time is affected mainly by Spark scheduling delays. This is tightly related to the cluster use tuning. This remains valid for target sample size (Table 4).

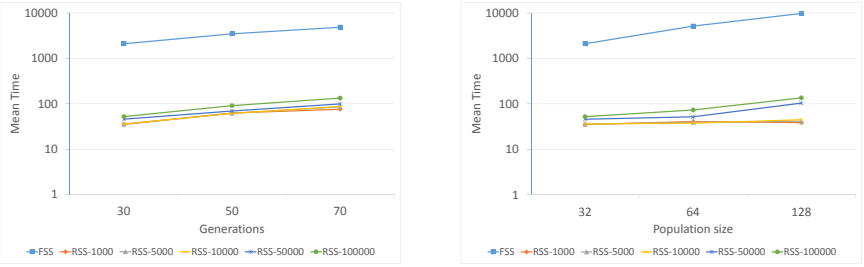
We tried the values: 1000, 5000, 10000, 50000 and 100000 instances per sample. While population size and generations number are set to 128 and 300.

To weigh the differences between using the whole dataset and a sampled subset

Table 4. RSS sample size effect on time.

Sample size	1000	5000	10000	50000	100000
Mean time (S)	384.748	411.198	547.839	971.921	1684.747

for training, we juxtapose the results of full dataset (FSS) with those obtained by RSS with the same number of generations and population size in Figure 6. Also, we keep an eye on learning performance in Figure 7. Undoubtedly, RSS outper-

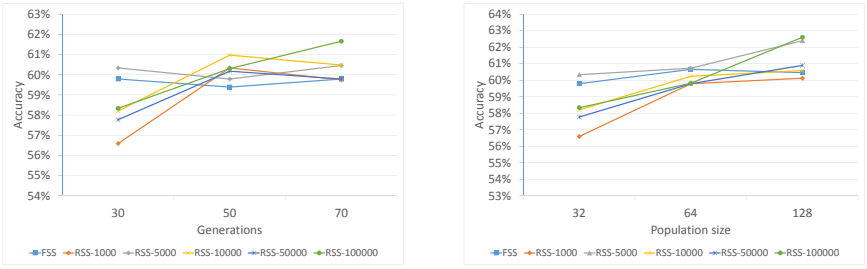


(a) Population Size = 32 (b) Generations = 30

Fig. 6. RSS vs FSS mean learning time.

forms GP without sampling for the 5 different target sizes. As regards learning performance, in terms of accuracy, experiments using RSS are less efficient with low number of generations or population size. It surpasses using the full dataset with large populations or more generations. On the one hand, in Figure 7(a), it's for more than 50 generations that all RSS variants have best accuracy. On

the other hand, Figure 7(b) shows that RSS outperforms the use of the entire dataset only with a population size of 128. It is important to notice that we



(a) Population Size = 32 (b) Generations = 30

Fig. 7. RSS vs FSS Best individual accuracy.

do not focus on enhancing learning performance and do not use any enhancing technique (e.g. feature engineering). Nevertheless, the best accuracy obtained is 66.93% with RSS (popultaion: 128, generations: 1500, sample: 10000) in 3190.02 seconds. The best result in [18] is 60.76% realized with logistic regression.

6 Conclusion

We presented, in this paper, details about reshaping DEAP library by parallelizing evaluation on Spark cluster. We obtained encouraging results that proclaim Spark as an efficient environment and is suitable for distributing GP evaluations. Then, we integrate a simple sampling technique that preserves learning performance while providing the possibility to probe GP with large populations or for a high number of generations. We studied experimentally the effect of varying 3 parameters: population size, generations number and sample size.

This work provides a Spark compliant GP implementation without the need to code it from scratch. Thus, it can be used in resolving different machine learning problems. Although it has been successfully tested on a small cluster, the size of the underlying cluster on the overall performance has to be investigated.

A logical extension is to study the impact of different Spark configurations (number of nodes, RDD partitioning, partition sizes, etc.). This will help to find the most suitable execution settings. A second path is to check the feasibility of adding other sampling techniques and in particular active sampling that prove to be advantageous in the context of machine learning. Hierarchical sampling, that we used in [11], is a promising candidate.

References

1. Al-Madi, N., Ludwig, S.A.: Scaling genetic programming for data classification using mapreduce methodology. In: Fifth World Congress on Nature and Biologically Inspired Computing, NaBIC 2013, August 12-14. pp. 132–139. IEEE (2013)

2. Baldi, P., Sadowski, P., Whiteson, D.: Searching for exotic particles in high-energy physics with deep learning. Nature communications **5** (2014)
3. Baldi, P., Sadowski, P., Whiteson, D.: Enhanced higgs boson to $\tau+$ τ- search with deep learning. Physical review letters **114**(11), 111–801 (2015)
4. Chávez, F., Fernández, F., Benavides, C., Lanza, D., Villegas-Cortez, J., Trujillo, L., Olague, G., Román, G.: ECJ+HADOOP: an easy way to deploy massive runs of evolutionary algorithms. In: Applications of Evolutionary Computation, EvoApplications 2016, March 30 - April 1, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9598, pp. 91–106. Springer (2016)
5. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: Brewer, E.A., Chen, P. (eds.) 6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004. pp. 137–150. USENIX Association (2004)
6. Fortin, F.A., De Rainville, F.M., Gardner, M.A., Parizeau, M., Gagné, C.: DEAP: Evolutionary algorithms made easy. Journal of Machine Learning Research **13**, 2171–2175 (jul 2012)
7. Funika, W., Koperek, P.: Scaling evolutionary programming with the use of apache spark. Computer Science (AGH) **17**(1), 69–82 (2016)
8. Gathercole, C., Ross, P.: Dynamic training subset selection for supervised learning in genetic programming. In: Parallel Problem Solving from Nature - PPSN III. Lecture Notes in Computer Science, vol. 866, pp. 312–321. Springer (1994)
9. Giráldez, R., Díaz-Díaz, N., Nepomuceno, I., Aguilar-Ruiz, J.S.: An approach to reduce the cost of evaluation in evolutionary learning. In: Cabestany, J., Prieto, A., Sandoval, F. (eds.) Computational Intelligence and Bioinspired Systems. pp. 804–811. Springer Berlin Heidelberg (2005)
10. Higgs Dataset: `http://archive.ics.uci.edu/ml/datasets/HIGGS`
11. Hmida, H., Hamida, S.B., Borgi, A., Rukoz, M.: Scale genetic programming for large data sets: Case of higgs bosons classification. Procedia Computer Science **126**, 302 – 311 (2018), the 22nd International Conference, KES-2018
12. Karau, H., Warren, R.: High Performance Spark. O'reilly, USA, 1st edn. (2017)
13. Kienzler, R.: Mastering Apache Spark 2.x. Packt Publishing (2017)
14. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)
15. Paduraru, C., Melemciuc, M., Stefanescu, A.: A distributed implementation using apache spark of a genetic algorithm applied to test data generation. In: Genetic and Evolutionary Computation Conference, July 15-19, Companion Material Proceedings. pp. 1857–1863. ACM (2017)
16. Peralta, D., del Río, S., Ramírez-Gallego, S., Triguero, I., Benitez, J.M., Herrera, F.: Evolutionary Feature Selection for Big Data Classification: A MapReduce Approach. Mathematical Problems in Engineering **2015**, 11 (2015)
17. Qi, R., Wang, Z., Li, S.: A parallel genetic algorithm based on spark for pairwise test suite generation. J. Comput. Sci. Technol. **31**(2), 417–427 (2016)
18. Shashidhara, B.M., Jain, S., Rao, V.D., Patil, N., Raghavendra, G.S.: Evaluation of machine learning frameworks on bank marketing and higgs datasets. In: 2nd International Conference on Advances in Computing and Communication Engineering. pp. 551–555 (2015)
19. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, April 25-27. pp. 15–28. USENIX Association (2012)